Analysis of the Windows Vista Security Model

Matthew Conover, *Principal Security Researcher, Symantec Corporation*

Abstract—**This paper provides an in-depth technical assessment of the security improvements implemented in Windows Vista, focusing primarily on the areas of User Account Protection and User Interface Privilege Isolation. This paper discusses these features and touches on several of their shortcomings. It then demonstrates how it is possible to combine these attacks to gain full control over the machine from low integrity, low privilege process.**

Index Terms—**Computer security, Windows Vista, Windows Resource Protection, File Virtualization, Registry Virtualization, Integrity Level, UAP, LUA, UIPI**

I. INTRODUCTION

W indows Vista is a radical departure from prior versions of the Windows operating system. With its introduction, enhancements have been made to virtually all aspects of the Windows security model. These changes should decrease the ease by which the operating system can be compromised.

In this research, Symantec researchers evaluated the security of the Windows Vista February 2006 CTP build. During this research we discovered a number of implementation flaws that continued to allow a full machine compromise to occur. By exploiting these flaws, a low privilege, low integrity level process can bypass User Account Protection, and ultimately execute code at a high privilege, high integrity level.

Since the conclusion of our initial phase of research, several new Windows Vista builds have been released. We recently re-evaluated our findings on the publicly released Windows Vista Beta 2 build 5384 and observed certain exploit paths have been fixed. Where applicable, we will indicate where our initial findings differ from the public Windows Vista Beta 2.

Windows Vista is a work in progress and it should be expected that security issues, including those discussed in this paper, will continue to be addressed until its final release.

A. What's Covered

This paper focuses on attacks against the Windows Vista security model from the perspective of malicious code. The scenario addressed in this paper is an out-of-the-box configuration that a typical user will see when presented with a new Windows Vista installation. In this configuration the user is a Protected Administrator [1] using Internet Explorer 7 to browse a malicious website that exploits a vulnerability [2]. This vulnerability inadvertently introduces malicious code running with low privileges on to the host. In this paper, we discuss a technique whereby a weakness in earlier Windows Vista builds will allow this malicious code to gain full control over the machine, ultimately acquiring `LocalSystem` privileges.

B. What's Not Covered

Malicious code that is already running with full `LocalSystem` privileges is outside of the scope of this paper, since the malicious code has roughly the same capabilities as it had in previous versions of Windows.

This paper only discusses the elevation of privileges to `LocalSystem`. Kernel-mode rootkits are also outside the scope of this paper. An assessment of Windows Vista kernel-mode security will be covered in a separate research paper. An assessment of the new Windows Vista TCP/IP network stack will also be covered in a separate research paper.

C. Prerequisites

As this paper is primarily focused on changes between Vista and the preceding Windows versions, the reader is expected to have familiarity with the traditional Windows security model—including general knowledge of Access Control Lists (ACLs), System ACLs (SACLs) versus Discretionary ACLs (DACLs), Security Identifiers (SIDs), etc. The reader is advised to review [4] and [5].

II. USER ACCOUNT PROTECTION (UAP)

A. Introduction

Windows Vista introduces a security feature, User Account Protection (UAP), which is also known as Least-Privilege User Accounts or Limited User Accounts (LUA). User accounts created during a Windows Vista installation are Protected Administrators and are subject to UAP. Protected Administrators are users in the Administrators groups, other than the Built-in Administrator (which is exempt from ever running under a LUA process). This means that when running without restriction, the user is capable of activities such as installing software, writing to `HKEY_LOCAL_MACHINE`, starting drivers, starting services, etc.

However, all processes launched by the Protected Administrator run with minimal privileges. When a Protected Administrator launches a program from the `Start Menu`, the program will run in a restricted context with a smaller subset of the privileges than the user actually possesses. If the

program requires administrative privileges (i.e., it won't function properly without them), the Protected Administrator can run the process unrestricted. By running the process unrestricted, the process inherits the full privileges of the user (referred to as *elevation*). A program will be run in an elevated state using one of the mechanisms discussed in Section III.A. Whenever a program is to be elevated, a popup box will appear asking the user to approve or deny. According to [1], there is no way to launch an elevated process from a Protected Administrator without the user's consent.

It is also possible to use a standard user account -- a user account without administrator privileges, rather than a Protected Administrator account. Standard user accounts were available in Windows XP. Although Microsoft recommends the use of standard user account, the default behavior when installing Windows XP or Windows Vista is to create an administrator user account. To create a standard user account, the user must perform additional manual steps. Therefore, we will only cover the default Windows Vista user account behavior that a general user would encounter after installing Windows Vista. This is not meant to imply that there are no privilege escalation attacks possible from a standard user account; rather, we focused our attention toward the most likely user configuration.

B. Mandatory Integrity Control (MIC)

Mandatory integrity control (referred to here as *integrity levels*) is a new feature added in Windows Vista. It is controlled by an access control entry (ACE) in the system access control list (SACL) of a securable object (e.g., a file, process, registry key, etc.). Every process will have an integrity level, and child (spawned) processes will inherit the integrity level from the parent process. Integrity levels can be enabled/disabled via the registry key `HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Policies\System\EnableMIC`.

Amusingly, the integrity level is associated with SACLs rather than DACLs (discretionary access control lists). DACLs are used to identify the trustee (a user, group, etc.) that are allowed or denied access to a securable object. In contrast, a SACL has had the historical role of generating audit records to record access to securable objects.

A process cannot interact with another process that has a higher integrity level. So `CreateRemoteThread`, `SetThreadContext`, `WriteProcessMemory`, and related APIs will fail from a lower integrity process when used against a higher integrity process. This is meant to prevent privilege escalation attacks. However it is still possible for:

1. A higher integrity process to call `CreateRemoteThread`, `SetThreadContext`, `WriteProcessMemory`, etc. against a lower integrity process.
2. Processes of any integrity level to interact using inter-process communication (named pipes, LPC, etc.).
3. A lower integrity server to impersonate a higher

integrity client using APIs such as `ImpersonateNamedPipeClient`, as long as the impersonation level of the client allows it.

Registry keys and files can have an integrity level as well; files or registry keys can only be written to if the process has a sufficient integrity level. This is why Low Rights Internet Explorer is only able to write to a small number of registry locations, even if it is run by a user who is a Protected Administrator.

The following table shows the integrity levels and their effective permissions:

Integrity Access Level	System Privileges
High	Administrative (can install files to the `Program Files` folder and write to sensitive registry areas like `HKEY_LOCAL_MACHINE`)
Medium	User (can create and modify files in the user's `Documents` folder and write to user-specific areas of the registry, such as `HKEY_CURRENT_USER`)
Low	Untrusted (can only write to low integrity locations, such as the `Temporary Internet Files\Low` folder or the `HKEY_CURRENT_USER\Software\LowRegistry` key)

The integrity access levels are governed by the following SACL ACEs:

SID	Integrity Level
S-1-16-16384	System Mandatory Level
S-1-16-12288	High Mandatory Level
S-1-16-8192	Medium Mandatory Level
S-1-16-4096	Low Mandatory Level

C. UI Privilege Isolation (UIPI)

Directly related to integrity levels is User Interface Privilege Isolation (UIPI), which was added to prevent privilege escalation attacks such as Shatter [6]. If a lower privileged process is able to send window messages (using the `SendMessage` and `PostMessage` APIs) to a higher privileged process, the lower privileged process can cause arbitrary code execution in the context of the higher privileged process. To address this, in Vista it is no longer possible for a process of a lower integrity level to send window messages to

a higher integrity process. This is enforced by the windowing and graphics subsystem known as USER (presumably within the system driver `win32k.sys`).

The `SetWindowsHookEx` and `SetWinEventHook` APIs provide a way to hook all other processes interacting with the same desktop and receive notification when those processes receive window messages. Internally, these APIs result in a DLL being loaded into all the other processes sharing the desktop. Prior to Windows Vista, this could also lead to arbitrary code execution in the context of a process with higher privileges than the initiating (calling) process had. In Windows Vista, lower privileged processes are prevented from invoking the `SetWindowsHookEx` and `SetWinEventHook` APIs on a higher integrity process.

UIPI is implemented at the process level. The process security descriptor will contain a special ACE in the SACL with SID S-1-16-16640 (UI Access Mandatory Level) to indicate the process is allowed to interact with more privileged processes sharing the desktop.

Certain processes, such as `uxss.exe` (Microsoft User Experience Subsystem) and `consent.exe` (Consent UI for administrative applications) are two processes that have the UI Access Mandatory Level, because they need to interact with the desktop.

D. Restricted Process

UAP is synonymous with *restricted process*. A restricted process is one with a restricted token that has some of the user's privileges removed and certain SIDs marked as "deny only" (see Appendix B). Restricted processes are setup using the `CreateRestrictedToken` API.

As an example, an unrestricted process created by a user in the Administrators group has the `SeDebugPrivilege`. This allows the user to debug any process on the system; it can be used as a way to manipulate and gain control over a more privileged process. Therefore, on Windows Vista, the majority of privileges are removed from a restricted process.

A restricted process created with UAP enabled has a reduced set of privileges:

```
SeChangeNotifyPrivilege          enabled
SeTimeZonePrivilege              disabled
SeIncreaseWorkingSetPrivilege    disabled
SeUndockPrivilege                disabled
SeShutdownPrivilege              disabled
```

By contrast, an unrestricted process created by an administrator has a much larger set of privileges:

```
SeChangeNotifyPrivilege          enabled
SeSecurityPrivilege              disabled
SeBackupPrivilege                disabled
SeRestorePrivilege               disabled
SeSystemtimePrivilege            disabled
SeShutdownPrivilege              disabled
SeRemoteShutdownPrivilege        disabled
SeTakeOwnershipPrivilege         disabled
```

```
SeDebugPrivilege                 disabled
SeSystemEnvironmentPrivilege     disabled
SeSystemProfilePrivilege         disabled
SeProfileSingleProcessPrivilege  disabled
SeIncreaseBasePriorityPrivilege  disabled
SeLoadDriverPrivilege            disabled
SeCreatePagefilePrivilege        disabled
SeIncreaseQuotaPrivilege         disabled
SeUndockPrivilege                disabled
SeManageVolumePrivilege          disabled
SeImpersonatePrivilege           enabled
SeCreateGlobalPrivilege          enabled
SeCreateSymbolicLinkPrivilege    disabled
SeIncreaseWorkingSetPrivilege    disabled
SeTimeZonePrivilege              disabled
```

If a privilege is disabled, it means it is ignored during access checks but can be enabled by the process. If a privilege is removed instead of disabled, as is the case for restricted processes, it cannot be enabled.

Another way in which a process is restricted is with the "Group used for deny only" attribute associated with the Administrators group SID. Normally, a process created by an administrator is granted access to most secured objects, because the Administrators group is explicitly granted access. For a restricted process, the Administrators group SID is set to "Group used for deny only." This means if the Administrators group is explicitly denied access, the restricted process is also denied access. On the other hand, if the Administrators group is explicitly granted access, this is *ignored* for restricted processes. Put another way, deny only SIDs match *access denied* rules but not *access granted* rules.

III.   UNRESTRICTED PROCESSES (ELEVATION)

A. Introduction

A process will be elevated under a few circumstances:
1. If the application is an installer (has the extension ".msi", matches a common installer like InstallShield, is named `setup.exe`, etc.)
2. Application Compatibility [7]
   a. If the application has an application compatibility entry in the registry under `HKEY_CURRENT_USER\Software\Microsoft\Windows NT\CurrentVersion\AppCompatFlags\Layers\<path_to_executable>` with the value `RUNASADMIN`
   b. AppCompat database entry (a file that ends with `<application_name>.sdb`) created with `CompatAdmin.exe`
3. The application's manifest [1] file (`<appname>.exe.manifest`) or resource (embedded within the executable) that contains `requestedExecutionLevel` of `requireAdministrator`
4. Manually by the user right-clicking on the executable and selecting "Run Elevated…" in Windows Explorer
5. Also when a program is:

a. Launched from an already privileged process
b. Launched from Task Manager via entering Ctrl-Shift-Esc or entering Ctrl-Alt-Del and clicking "Task Manager" (see the next section for more details). This elevation trick has since been resolved in more recent Windows Vista builds.

COM objects can also run elevated in the form of out-of-process DLLs loaded into a privileged surrogate process. This occurs when in `HKEY_CURRENT_ROOT\Classes\CLSID\<CLSID>\Elevation\Enabled` is set to 1. When this happens, a Consent Popup will ask for the user's authorization. If the user approves, the COM object is elevated via the `CoCreateInstanceAsAdmin` API [1], which is new to Windows Vista. The `CoCreateInstanceAsAdmin` API will initiate an RPC call to the `AppInfo Admin Broker` service, which will load the DLL in a child process of `svchost.exe` running as `LocalSystem` (either `rundll32.exe` or another instance of `svchost.exe`). The following COM objects are currently configured to be run elevated in the registry:

{08d450b7-f7e5-4424-8229-11888adb7c14}	%SystemRoot%\system32\fontext.dll
{1138506a-b949-46a7-b6c0-ee26499fdeaf}	%SystemRoot%\system32\wucltux.dll
{26FE7361-BD5A-4DCB-B309-C6F42DDE661C}	"%ProgramFiles%\Internet Explorer\IEInstal.exe"
{304CE942-6E39-40D8-943A-B913C40C9CD4}	c:\Windows\system32\wfapi.dll HNetCfg.FwMgr
{33E5987B-CA8A-4a8a-921A-8AC16A1676EB}	%SystemRoot%\System32\shpafact.dll
{375C3A49-8654-49C6-BD32-7E7FE88509B4}	%programfiles%\AdhocMeetings\WinCollabElev.dll WinCollabElev.Elev.1
{3ad05575-8857-4850-9277-11b85bdb8e09}	%SystemRoot%\system32\shell32.dll
{49F371E1-8C5C-4d9c-9A3B-54A6827F513C}	ntshrui.dll
{4BC67F23-D805-4384-BCA3-6F1EDFF50E2C}	c:\Windows\system32\wercplsupport.dll ERCLuaElevationHelper
{514B5E31-5596-422F-BE58-D804464683B5}	intl.cpl
{6311429E-2F1A-4777-880F-C7289FD10169}	ntshrui.dll
{7007ACD1-3202-11D1-AAD2-00805FC1270E}	%SystemRoot%\System32\netshell.dll
{71B804C5-5577-471D-8FE5-C4A45B654EB8}	%SystemRoot%\System32\AuxiliaryDisplayCpl.dll
{72A7994A-3092-4054-B6BE-08FF81AEEFFC}	%SystemRoot%\System32\shpafact.dll
{77F419AA-771A-45ff-AC66-7567FA3243D3}	ntshrui.dll
{86d5eb8a-859f-4c7b-a76b-2bd819b7a850}	%SystemRoot%\System32\shpafact.dll
{8c2db90a-6c3d-48fa-a571-0be2836c630c}	%SystemRoot%\System32\shpafact.dll
{9df523b0-a6c0-4ea9-b5f1-f4565c3ac8b8}	timedate.cpl
{a036417d-768d-4566-8be4-5f5e1268fa9f}	%SystemRoot%\System32\ntshrui.dll
{A0ADD4EC-5BD3-4f70-A47B-07797A45C635}	%SystemRoot%\System32\cscui.dll
{A2D75874-6750-4931-94C1-C99D3BC9D0C7}	%ProgramFiles%\Windows Defender\MsMpCom.dll
{A3BB0AD5-ECA3-4A81-B2CB-15FD8349D400}	%SystemRoot%\System32\SLLUA.exe SLLUA.SLLUAObject.1
{A7A63E5C-3877-4840-8727-C1EA9D7A4D50}	%SystemRoot%\System32\fveui.dll
{afb8cfa2-6d7b-4108-9202-cc08d7222dc9}	%SystemRoot%\system32\shell32.dll
{bCEA735B-4DAC-4B71-9C47-1D560AFD2A9B}	DfsShlEx.dll DfsShell.DfsShellAdmin.1
{c529C7EF-A3AF-45F2-8A47-767B33AA5CC0}	%SystemRoot%\system32\ndfapi.dll ndfapi.NDFAPI.1
{cee8ccc9-4f6b-4469-a235-5a22869eef03}	PNPXAssoc.dll
{D3667F1E-CCB8-4A69-99DF-59A2B2A6753F}	%SystemRoot%\System32\AuxiliaryDisplayCpl.dll
{e6f59608-8aa2-4dbe-a651-c2f6585e4f30}	%SystemRoot%\System32\shpafact.dll
{E9495B87-D950-4ab5-87A5-FF6D70BF3E90}	wscui.cpl
{edb5f444-cb8d-445a-a523-ec5ab6ea33c7}	ntshrui.dll

B. The Legacy Shell Trick

The elevation trick using Task Manager mentioned in the previous section (case 5b) has since been resolved in more recent Windows Vista builds. It originally worked because the process is launched from `WinLogon`. This occurs because `WinLogon` is responsible for handling the Secure Attention Sequence, which is what Ctrl-Alt-Del and Ctrl-Shift-Esc are. `WinLogon` runs unrestricted and with high integrity. An executable that is launched from here via the "File -> New Task" menu option of Task Manager runs with full privileges. This does not seem to have been intentional. In fact, [1] states, "The only way for the Shell to run as administrator is to log on with the machine administrator account (Built-in Administrator)." This statement is inaccurate. It was possible to kill the existing `Explorer.exe` from Task Manager and restart it via "File -> New Task" menu option, and entering "explorer". Task Manager launches processes via `CreateProcess` instead of `CreateRestrictedProcess`, so Windows Explorer is launched without restrictions and operates like the legacy shell from Windows XP. That is, there will no longer be any consent prompts when launching applications, and files can be moved, renamed, deleted without needing to elevate.

C. Consent Prompts and Admin Brokers

Since Windows Explorer itself runs with a restricted token and medium integrity level, it lacks sufficient privilege to launch the application unrestricted on its own. Instead, it uses a surrogate: the `AppInfo Admin Broker` service. `AppInfo` runs as `LocalSystem` and has more privileges than even a Protected Administrator. Because it is a service, `AppInfo` runs in Isolated Session 0 (discussed in the next section).

The `AppInfo Admin Broker` service exposes an RPC interface function `RunAsAdminProcess`. When a process is to be run elevated, Windows Explorer (through the `ShellExecute` API) uses this RPC interface to request `AppInfo` launch the application with the user's full credentials.

When `AppInfo` receives the RPC request, it launches `consent.exe` which produces a popup window on the

desktop asking for the user's consent. If the user clicks "Approve", `AppInfo` calls the `ImpersonateLoggedOnUser` and `CreateProcessAsUser` APIs to launch the process as the user. It is necessary for `AppInfo` to use `CreateProcessAsUser` instead of `CreateProcess`, because the `AppInfo` service is used by all users and there may be more than one user logged onto the machine.

IV. SERVICE ISOLATION ("ISOLATED SESSION 0")

In previous versions of Windows, services and user processes all ran under the same session. With Vista, services run in the "Isolated Session 0" [8]. This means that a normal service cannot show any popup or dialog boxes to the user. If a service tries to generate a popup event or a dialog box to receive user interaction, it will sit forever since the user will not be able to see it. This is true *even if* the service is configured as interactive (i.e., allowed to interact with the desktop). The interactive session the user logs into is, in fact, a Terminal Server session. Using `query.exe`, one can see all the interactive sessions. In fact, the Microsoft-sanctioned way to send a message from a service running in the Isolated Session 0 is to use `WTSSendMessage` which is part of the Windows Terminal Services API.

How does `AppInfo` running under a surrogate `svchost.exe` process in the Isolated Session 0 cause the Consent Prompt in another session? It uses `CreateProcessAsUser` to launch `consent.exe` in Session 1 with the "UI Access Mandatory Level" discussed in Section II.C supra.

V. FILE AND REGISTRY VIRTUALIZATION

A. Introduction

Windows developers have usually assumed the user has administrative privileges. Operations such as writing to the Windows system directory or the `HKEY_LOCAL_MACHINE` registry require administrative privileges; a standard user (likewise, under a LUA process) cannot write to these locations. According to [1], disabling virtualization *"... will result in a regression to a Windows XP application compatibility pass rate of 56 percent."*

To work around this, Microsoft has introduced file and registry virtualization to retain applications backwards-compatibility. When lower privileged processes that attempt to modify global locations fail due to lack of permission, the data is instead transparently written to a per-user location (known as *virtualization*). These per-user locations are checked before global locations. In other words, the per-user location overrides the global location.

B. Registry Virtualization

Registry virtualization is implemented by `ntoskrnl` and `ntkrnlpa` (i.e., the operating system kernel itself). When running under a LUA process, registry write attempts that fail (due to insufficient permission) have their location changed from:
```
HKEY_LOCAL_MACHINE\Software
```
to:
```
HKEY_CURRENT_USER\Software\Classes\VirtualSto
re\MACHINE\Software
```

C. File Virtualization

File virtualization is implemented by the file system filter driver `luafv.sys`. When running under a LUA process, file write attempts that fail (due to insufficient permission) have their location changed from:
```
C:\Progra~1 (C:\Program Files)
```
to:
```
%UserProfile%\AppData\Local\VirtualStore\C\P
rogra~1
```

For example, if a LUA process tries to replace a configuration file (e.g., `%WinDir%\win.ini`) and lacks sufficient privileges to modify the real `%WinDir%\win.ini`, then `win.ini` is virtualized to the per-user location. If that user later reads from `%WinDir%\win.ini`, the user will see his/her modifications. However, no other users will see these modifications.

It is also interesting to note that certain executable file extensions (`cmd`, `bat`, `exe`, `dll`, etc.) are not virtualized, even though this isn't mentioned in [9]. This means the user cannot overwrite an executable file by file virtualization. Attempts to do so will fail due to access denied. For a complete list, see Appendix F at the end of this paper.

D. Low Rights Internet Explorer Virtualization

Virtualization for Low Rights Internet Explorer is not done by the file system driver that handles normal file and registry virtualization. Instead this is done by an AppCompat shim DLL located in `%WinDir%\AppCompat\iebrshim.dll` (IE Broker Shim). A low integrity process cannot even write to the user-specific locations used for LUA file and registry virtualization. That is, a low integrity process cannot even write to `%UserProfile%\AppData\Local\VirtualStore` used for medium integrity level file virtualization or `HKEY_CURRENT_USER\Software\Classes\Virtual` used for medium integrity level registry virtualization. This is to prevent low-to-medium privilege escalation attacks, as LUA processes run at the medium integrity level.

Since all files and registry keys have a default integrity level of medium, a low integrity process is only able to write to locations that have been explicitly allowed (by setting the integrity level to low integrity).

Per-user file virtualization is in:
```
%UserProfile%\AppData\Local\Microsoft\Window
s\Temporary Internet Files\Virtualized
```

Per-user registry virtualization is in:

```
HKEY_CURRENT_USER\Software\Microsoft\Interne
t Explorer\InternetRegistry
```

Other low integrity file locations are:
- `%UserProfile%\AppData\Local\Microsoft\ Windows\Temporary Interface Files\Low`
- `%UserProfile%\AppData\Local\Microsoft\ Windows\History\Low`
- `%UserProfile%\AppData\Local\Temp\Low`

Other low integrity registry locations are:
- `HKEY_CURRENT_USER\Software\Microsoft\I nternet Explorer\LowRegistry`
- `HKEY_CURRENT_USER\Software\Microsoft\W indows\CurrentVersion\Internet Settings\5.0\LowCache`
- `HKEY_CURRENT_USER\Software\Microsoft\W indows\CurrentVersion\Explorer\MenuOrd er\Favorites`
- `HKEY_CURRENT_USER\Software\Microsoft\I nternet Explorer\Toolbar`

Most settings are controlled by `HKLM\Software\Microsoft\Internet Explorer\Low Rights`.

When the user launches Internet Explorer, it will first run `IEUser.exe`, which runs at the default medium integrity level. `IEUser.exe` will spawn a low-integrity `IExplore.exe`, which is what the user sees and interacts with. When the user wishes to install an ActiveX control or something requiring administrative privileges, the `IEInstall.exe` Admin Broker is used (which runs at high integrity). `IEInstall.exe` is marked as an elevated `LocalServer32` COM object (CLSID 26FE7361-BD5A-4DCB-B309-C6F42DDE661C). When the user wishes to save a downloaded file, this will be done by `IEUser.exe` which runs with medium integrity. Here is a diagram from [10] which illustrates this:
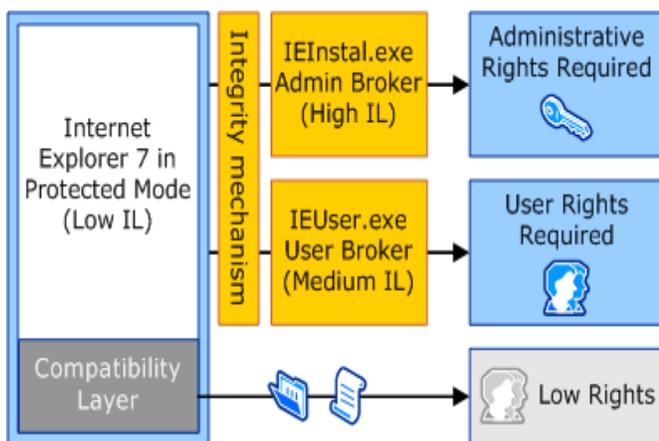
VI. WINDOWS RESOURCE PROTECTION (WRP)

Windows Resource Protection (WRP) replaces the System File Protection (SFP) that existed in previous versions of Windows. WRP now protects more than just files, it also protects registry keys. All signed executable code and drivers in an OS manifest are protected by WRP.

SFP worked by registering for notification of file changes in `WinLogon`. If any changes were detected to a protected system file, the modified file was restored to a saved copy located in `%WinDir%\System32\dllcache`.

In Windows Vista, WRP works by setting the ACLs on the protected file and registry keys so that they can only be written to by the processes with the `TrustedInstaller` SID. `LocalSystem` and Administrators are given read-only access. The result is that the files and registry keys can only be modified by the `TrustedInstaller` service, which is located in `%WinDir%\servicing\TrustedInstaller.exe`.

Windows Update will call the `TrustedInstaller` to make OS updates. The `TrustedInstaller` will only install signed updates.

VII. ATTACKS

A. Introduction

Windows Vista has unique challenges in trying to prevent privilege escalation attacks compared to the approach taken by UNIX derivatives. This section will focus on the privilege escalation attacks that result from the approach Microsoft has taken with Windows Vista.

If a higher integrity process uses registry keys and configuration files that are writable by a lower integrity process, then the security model is tainted, as this permits a lower integrity process to influence the behavior of a higher integrity process. In this section, we will show a number of flaws in the LUA implementation and, as a result, show how it allows privilege escalation from low integrity to medium integrity, then medium integrity to high integrity, then high integrity to `LocalSystem`.

B. UNIX Security Model

UNIX has supported standard user accounts with limited privileges for decades. Therefore, UNIX programmers are well adjusted to accommodating standard users. If a limited user wants to install a program into a global location that the user doesn't have write access to, the user will need to `su` (the switch user command) to a user that has write access to the global location (usually the root account). The limited user accounts also serve as a form of sandboxing. For example, it is common for the Apache web server to run under the user account `apache`. File permissions can be set to prevent `apache` from reading/writing to anything the web server doesn't need access to. Then if the web server is compromised, the attacker is restricted by the limited access of the `apache` account.

We have over-generalized this to avoid having to discuss specifics (Linux Security Modules, chroot, etc.). Some UNIX security models are quite similar to Windows Vista. For example, SELinux also has mandatory access controls and per-process privilege levels that are fixed at the time of program execution.

C. Windows Visa Security Model

Windows Vista's developers had to choose the best way to improve the overall security model while still retaining the most backward compatibility. While most of their decisions seem reasonable, two particular decisions lead to several seemingly intractable implementation flaws.

First, Windows programmers have been quite lax on leveraging and exercising rights and privileges in the existing Windows security model. A common behavior is to open a registry key or file for all access, when really only read access was needed. Another problem is making the assumption the user has administrative privileges (and requiring more privileges than actually needed). These assumptions are not just made by third-party Windows programmers—several Microsoft-implemented programs also fail without administrative privileges (e.g., the clock in the taskbar and `shutdown.exe`). For this reason, Microsoft has been forced to use "Application Compatibility" shims and file/registry virtualization to allow pre-Vista programs to function properly.

Second, Windows Vista can have several processes created by the same user operating at different integrity levels. This obviously creates an incentive for a low integrity level process to try to acquire the higher integrity level of the other process created by the same user. Windows Vista tries to close to obvious holes: for example, UIPI to prevent a lower integrity process from sending window messages to a higher integrity process. There is still far too much overlap between processes running at different integrity levels. A medium integrity processes can modify registry keys under `HKEY_CURRENT_USER` which are also used by high integrity processes.

D. From Low Integrity Level

Scenario:

A user browses a malicious website using Low Rights Internet Explorer 7. The malicious website then exploits a previously unknown vulnerability, resulting in arbitrary code execution in the context of `IExplore.exe` (running at the low integrity level). The goal of the malicious code at this stage is to obtain execution on the target and reach medium integrity level.

Internet Explorer is the only process running at low integrity on a default Windows Vista installation. At the low integrity level, Internet Explorer is running below the default integrity level of other services. On the February build, we discovered that a low integrity process that reconnects to the same machine over the network (i.e., loops back) via SMB can again enjoy slightly elevated privileges.

It should be noted that the vulnerability we will now discuss was closed as of the public Windows Beta 2 (build 5384), however it is instructive to observe how it works since other services (e.g., IIS, PNP, IPv4 over IPv6) might offer a malicious program other opportunities to "loop-back".

If the user account is "`matt`" and the user's home directory is `c:\users\matt`, a medium integrity process can create `c:\users\matt\test.txt`, whereas the low integrity process cannot (since the default integrity level of a file/folder is always medium integrity). So, if a low integrity process can force the use of the user's default credentials, it results in a privilege escalation. Thus, if a malicious ActiveX object tries to write to "`c:\users\matt\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup\malicious.exe`" in order to gain execution when the next time the user logs in, it will fail. *However*, a malicious ActiveX object *can loopback to the directory using SMB and* write to "`\\127.0.0.1\C$\users\matt\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup\malicious.exe`." This is because a write operation through a file share will use the default integrity level of medium. This share is enabled by default and normally only available to Administrators. Since the attacker is connecting to a local share, no username/password is needed by the malicious code. The primary ways to exploit this vulnerability are to:

1. Place a malicious executable in the `Startup` folder. This is the most straightforward attack, but the malicious program will not be run until the user's next login.
2. Replace a program shortcut the Start Menu such as `Internet Explorer.lnk` so that when the current user clicks the Internet Explorer button, it launches a malicious program at medium integrity level. This is seemingly the most effective attack since it is quite unlikely the user will notice the change.

E. From Medium Integrity Level - Introduction

Scenario:

The low integrity escalation vulnerability discussed in the previous section was used to place "`malicious.exe`" in the user's `Startup` folder. Programs in this folder are executed when the user logs on to the machine. So for this section, it is assumed the user logged off and later logged back into the machine, resulting in the execution of "`malicious.exe`". Thus "`malicious.exe`" is now executing at the medium integrity level.

To elevate privileges from medium to high integrity level, it is necessary to find a high integrity process that can be influenced by a medium integrity process. Possible attacks are:

- Communicate over named pipe or LPC with a high integrity process (e.g., `CSRSS`, `LSM`, `SMSS`, etc.) and try to get the high integrity process to, for example, write to a file that a medium integrity process cannot.
- Find a shared memory section used by a high integrity process that is writable with medium integrity level, similar to the technique used in [11].
- Find a configuration file or registry key that is writable from a medium integrity process and used as input in a high integrity process.

Specifically, everything under `HKEY_CURRENT_USER` is writable from a medium integrity process, and high integrity processes also have a high degree of interaction with `HKEY_CURRENT_USER`. This is a logical vector for attack and potentially creates a large problem.

F. From Medium Integrity Level – Method 1

In this attack, a subterfuge (i.e., slight of hand trick) is used to mislead the user. There are certain operations performed by the user where the user expects to see a consent prompt asking for authorization to elevate a process. Microsoft does *not* lock the executable in question prior to prompting for consent. So if this executable file being elevated is in a location writable at medium integrity level, then it is possible to replace the file after the consent prompt but before the user clicks "Approve." For example, the user attempts to run a trusted program that is signed by Microsoft, so the consent prompt will have a green banner to show it is signed by a trusted vendor. The attack works by having `malicious.exe` running in the background at medium integrity. Whenever `consent.exe` is launched, the malicious program will check which program is being launched and see if it has write access. If it does, it will copy its own malicious application over the program that is about to elevated. The user reaction time (especially at today's processor speeds) to see the consent prompt and click "Approve" provides ample time to overwrite the program about to be launched. So the user will end up launching a malicious program, thinking that they were running a signed, trusted program as the consent prompt indicated. The user will have no way to detect to this.

COM objects are frequently represented as DLLs. So when a process wants to load the COM object, it is loaded using rundll32.exe to load the COM object into a surrogate process like svchost.exe or dllhost.exe. If the COM object requires administrative privileges, then `CoCreateInstanceAsAdmin` is called and the user is prompted for consent. Presumably, the DLL of the COM object can also be overwritten in the same manner as executables can. If so, then the user can also be misled during elevation of COM objects. At the time of publication, this conjecture has not been experimentally verified.

G. From Medium Integrity Level – Method 2

This attack also uses a subterfuge to mislead the user. Global COM objects are located under `HKEY_LOCAL_MACHINE\Software\Classes\CLSID`. User-specific COM objects are located under `HKEY_CURRENT_USER\Software\Classes\CLSID`. Because COM objects under `HKEY_CURRENT_USER` take precedence over COM objects in `HKEY_LOCAL_MACHINE`, "`malicious.exe`" can enumerate all elevated COM objects under `HKEY_LOCAL_MACHINE`, and re-create the same COM entries under `HKEY_CURRENT_USER`, and substitute the values with the paths to malicious COM objects. Users may be easily deceived by this subterfuge and unwittingly consent to execute the malicious COM objects. For example, most of the items under the control panel are COM objects that will result in consent prompts. At the time of publication, this conjecture has not been verified.

If this is the only step implemented, then a keen user may notice that the consent prompt is red or yellow (unsigned or untrusted) instead of green (signed and trusted). Windows considers an application trusted if it is signed by a trusted certificate authority with Authenticode.

While the attack described in this section was found to exist in the February CTP, it has been fixed as of the public Windows Beta 2 (build 5384). Later Vista builds only refer to `HKEY_LOCAL_MACHINE` registry keys for COM elevation and ignore entries under `HKEY_CURRENT_USER` so this exploit path is no longer possible.

H. From Medium Integrity Level – Method 3

The final attack makes use of the `AppInit_DLLs` key and requires registry virtualization to be enabled. This attack is only effective against the 32-bit version of Windows Vista, as registry virtualization is disabled on the 64-bit version of Windows Vista. This key is located under `HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Windows`. To implement this attack:

- Create a `REG_SZ` entry named `AppInit_DLLs` that and set it to the full path to the malicious DLL
- Create a `REG_DWORD` entry named `LoadAppInit_DLLs` set to `1`

A medium integrity process *cannot* write to HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Windows. Instead, it will be redirected (virtualized) to HKEY_CURRENT_USER\Software\Classes\VirtualStore\MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Windows. Because the virtualized `AppInit_DLLs` takes precedence, the malicious DLL specified in the virtualized `AppInit_DLLs` will be loaded into all processes running under UAP. Processes running with full administrative privileges (i.e., non-UAP elevated processes) are unaffected, because registry virtualization doesn't apply to them.

In most cases this would not result in any elevation, because high integrity processes are almost never running under LUA, and thus don't make use of the malicious `AppInit_DLLs` in the registry VirtualStore. However, `uxss.exe`, the User Experience Subsystem, is an exception. It is a high integrity process and also has the UI Access Mandatory Level. This is necessary because `uxss.exe` controls the desktop, so it is responsible for sending window messages to all GUI programs that interact with the desktop. `Uxss.exe` also needs to run at high integrity in order to send messages to other high integrity processes that may interact with the desktop (such as `consent.exe` which runs as `LocalSystem`). Unfortunately, this permits `uxss.exe` to conduct Shatter attacks against

consent.exe.

Making uxss.exe a restricted process running under LUA actually makes things worse. Because it runs under LUA, it is subjected to registry virtualization, and yet at the same time runs at high integrity with UI access. The ultimate result is that when the user logs in, uxss.exe is launched as a high integrity LUA process, reads the virtualized AppInit_DLLs, and loads a malicious DLL into the high integrity (but restricted) uxss.exe process.

As of the public Windows Beta 2 (build 5384), uxss.exe does not exist and since that was the only exploitable process that was discovered, this exploit path does not work any longer. Furthermore, registry virtualization is now configurable on a per process basis.

I. From High Integrity Level, LUA Process

Scenario:

The attack discussed in the previous section was used by malicious.exe to prepare its malicious.dll to be loaded into uxss.exe. Once the user logged off and logged back on, "malicious.dll" was loaded into the address space of uxss.exe, a high integrity but restricted process.

At this stage, Vista is a step away from becoming checkmated and full compromise is trivial. Microsoft ignored its own advice, stated in [12]:

"*Applications that use restricted tokens should run the restricted application on desktops other than the default desktop. This is necessary to prevent an attack by a restricted application, using SendMessage or PostMessage, to unrestricted applications on the default desktop. If necessary, switch between desktops for your application purposes.*"

Since malicious.dll has high integrity, the SetWinEvent and SetWindowsHookEx APIs can be used. These allow a DLL to be launched into all processes interacting with the same desktop. So all malicious.dll has to do is use one of these APIs and wait for a high integrity, unrestricted process to be launched. While an effort was made to reduce the interaction of high privilege services with the user's desktop (by placing them in an isolated session as discussed previously), it was not possible to eliminate the interaction entirely. Namely, csrss.exe, ctfmon.exe, LogonUI.exe, WinLogon.exe, and consent.exe are all processes running with LocalSystem capabilities within Session 1. So, as soon as one of these processes interacts with the desktop, the module specified in the hModule parameter of SetWinEvent or SetWindowsHookEx will be loaded and its DllMain will be called.

While the attack described in this section was found to exist in the February CTP, it has been fixed as of the public Windows Beta 2 (build 5384). There is no longer an uxss.exe in the latest Vista builds and it was the only process known to be exploitable.

J. Against Windows Resource Protection

Scenario:

The attack in Section I was used and now malicious.dll is running as LocalSystem.

The following attack could be done from either LocalSystem or any account in the Administrators group, as long as it is a non-LUA process (since the SeTakeOwnership privilege is needed). LocalSystem and Administrators both have the ability to take ownership of files. Windows Resource Protection, as mentioned previously, is implemented as an ACL that only grants write access to the TrustedInstaller SID. However, because Administrators and LocalSystem both have sufficient privilege to take ownership of securable objects, the steps to evade WRP are to first enable the SeTakeOwnership privilege, second take ownership of the WRP-protected file or registry key, and finally grant Administrators full access. These steps can be done using the AdjustTokenPrivileges (for step 1) and SetNamedSecurityInfo (for steps 2 and 3) APIs. After that, the WRP-protected file or registry key can be changed without inhibition. There is no longer a thread that attempts to detect changes to protected system files as was done by SFP prior to Windows Vista. Therefore, it possible to backdoor all system files at this stage. In addition, driver signing restrictions will not help to mitigate this attack. We have successfully demonstrated in a second paper, "Assessment of Windows Vista Kernel-Mode Security," that driver signing restrictions can be disabled with two binary patches: one to WINLOAD.EXE and one to CI.DLL.

K. Failed Attacks

This section includes attacks that were unsuccessful. In some cases, the attack scenario was thoroughly tested and Windows Vista seems to properly defend against it. Conducting blackbox testing as we did, there always exists the possibility of a false negative – a probe that succeeds with a subtle side-effect we did not notice, may eventually be used to mount a successful attack upon the operating system's security perimeter. This section will highlight the areas of Windows Vista where attacks were unsuccessful:

Silent installs do not result in any silent elevation. Instead, a silent install runs with the credentials of the user and the install fails if more privileges are later required, without prompting the user.

The attack discussed in Section VII.D allows a low integrity process to write to any file that a medium integrity process can. The goal of Section VII.D was to find a way to run an arbitrary application with medium integrity. The method chosen was to place an executable in the Startup folder in the user's Start Menu. An alternative method, that didn't work in testing, would be to place a malicious desktop.ini in a folder likely to be browsed Windows Explorer. Here is a sample desktop.ini:

```
[.ShellClassInfo]
IconFile=%SystemRoot%\system32\shell32.dll
IconIndex=-173
LocalizedResourceName=@shell32.dll,-12693
```

Windows Explorer checks for the presence of desktop.ini when browsing a folder to allow per-folder customization. A good candidate is `%AppData%\Microsoft\Windows\Start Menu\desktop.ini` (this would be checked each time the user clicked the Start Menu). The theory was that by giving the path to a malicious DLL in the `desktop.ini`, Windows Explorer would load this DLL when that directory is browsed. However, during experimentation, this turned out not to be the case. The DLL referenced in the malicious `desktop.ini` was never loaded. It may be worth later investigating why this didn't work. Presumably the attack didn't work because the DLL is loaded as a data file and only checked for its resource section (rather than loaded as an executable image). For the attack to work, the DLL needs to be loaded as a conventional DLL using the `LoadLibrary` API, which would cause the DLL to execute and thereby introduce malicious code into Windows Explorer.

While [1] states that sending window messages from a lower integrity process to a higher integrity process is prevented, it didn't mention whether the GUI-related APIs `SetWindowsHookEx` and `SetWinEventHook` are also prevented. However, it has been verified that these are also prevented from a lower integrity process to a higher integrity process.

Attempts to override an existing executable such as `%WinDir%\system32\calc.exe` by placing a malicious `calc.exe` in the corresponding `VirtualStore` location failed. It was later determined that this is due to Windows Vista excluding certain file extensions (see Appendix F) from virtualization. This seems to be undocumented in all of the Microsoft documents mentioning file virtualization, however this is clearly revealed by analyzing the file system filter driver that implements file virtualization (`luafv.sys`),

VIII. CONCLUSION

Although we discovered several weaknesses in Windows Vista Feburary 2006 build, later builds have corrected the implementations and closed the exploit paths. Windows Vista's out-of-the-box security is a significant improvement over previous versions of Windows. It is likely that the security community will aggressively probe and seek to undermine Vista's security improvements once it is released. The author expects several other privilege escalation vulnerabilities to be discovered. There are two areas we can expect to be scrutinized heavily by malicious code authors and Spyware vendors trying to work around the additional security restrictions:

- *Ways to acquire medium or high integrity from low integrity*. This is of great interest to malicious code authors and Spyware vendors trying to break out of the Low Rights Internet Explorer sandbox.

- *LPC/RPC interfaces exposed from high integrity processes*. Since a process of any integrity level can send commands to these server interfaces, it can provide an unintended method for a client to perform more privileged operations than it should. To illustrate, consider the `AppInfo Admin Broker` discussed in Section III.C. It runs at high integrity and exposes an RPC function `RunAsAdminProcess`. This can be called from a low integrity level. Normally this is not a problem, because `AppInfo` prompts the user for consent. If `AppInfo` had a vulnerability (perhaps an undocumented flag) that would cause `AppInfo` to execute the command the low integrity client supplied without a Consent Prompt, this would provide a trivial way for a malicious process to acquire higher privileges. There are number of RPC interfaces exposed from high integrity processes, so it is just a matter of an attacker finding one that can be abused. The author fully anticipates this will happen.

Because Windows Vista is still in beta, some of the behavior described may change prior to Windows Vista's public release. Therefore, it is advised the reader continues to follow up on Microsoft Vista blogs such as [14] and [15].

REFERENCES

[1] Microsoft. (2005, September). Developer Best Practices and Guidelines for Applications in a Least Privileged Environment. *MSDN* [Online]. Available: http://msdn.microsoft.com/windowsvista/default.aspx?pull=/library/en-us/dnlong/html/AccProtVista.asp

[2] Slashdot. First Windows Vista Security Update Released [Online]. Available: http://it.slashdot.org/article.pl?sid=06/01/15/1910205

[3] T. Newsham. Windows Vista Networking: A Broad Overview [Online]. Available: http://www.symantec.com/avcenter/reference/ATR-VistaAttackSurface.pdf

[4] Microsoft. Access Control. *MSDN* [Online]. Available: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/secauthz/security/access_control.asp

[5] M. Russinovich, D. Solomon. *Microsoft Windows Internals, Fourth Edition: Microsoft Windows™ 2003, Windows XP, and Windows 2000*. Redmond, WA: Microsoft Press, 2005, ch 8.

[6] B. Moore. (2003, October). Shattering by Example [Online]. Available: http://www.security-assessment.com/Whitepapers/Shattering_By_Example-V1_03102003.pdf

[7] Microsoft. Using Application Compatibility Tools for Marking Legacy Applications with Elevated Run Levels on Microsoft Windows Vista. *MSDN* [Online]. Available: http://www.microsoft.com/technet/windowsvista/deploy/appcompat/acshims.mspx

[8] A. Ben-Menahem, A. Tucker. "Windows Vista and 'Longhorn' Server: Understanding, Enhancing and Extending Security End-to-end," PDC, Los Angeles, 2005. Available: http://microsoft.sitestream.com/PDC05/FUN/FUN210_files/Botto_files/FUN210_Ben-Menahem_Tucker.ppt

[9] R. B. Ward, K. Thirumalai. "Windows Vista and 'Longhorn' Server: Under the Hood of the Operating System Internals and Your Application," PDC, Los Angeles, CA, 2005. Available: http://microsoft.sitestream.com/PDC05/FUN/FUN417_files/Botto_files/FUN417_Ward_Thirumalai.ppt

[10] Microsoft (2006, January). Understanding and Working in Protected Mode Internet Explorer [Online]. Available: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/IETechCol/dnwebgen/ProtectedMode.asp

[11] C. Cerrudo. "Hacking Windows Internals," Blackhat Europe, Amsterdam, Netherlands, 2005. Available: http://www.blackhat.com/presentations/bh-europe-05/BH_EU_05-Cerrudo/BH_EU_05_Cerrudo.pdf

[12] Microsoft. Restricted Tokens. *MSDN* [Online]. Available: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/secauthz/security/restricted_tokens.asp

[13] Skape, skywing. (2005, December, 1). Bypassing PatchGuard on Windows x64 [Online]. Uninformed (Volume 3). Available: http://www.uninformed.org/?v=3&a=3&t=txt

[14] Microsoft. *UACBlog* [Online]. Available: http://blogs.msdn.com/uac

[15] Microsoft. *IEBlog* [Online]. Available: http://blogs.msdn.com/ie

[16] M. Conover. Malware Profiling and Rootkit Detection on Windows [Online]. Available : http://www.cybertech.net/~sh0ksh0k/projects/ObjProfiler

APPENDIX

A. Protected Administrator, Low Rights Internet Explorer

USER INFORMATION

```
User Name    SID
hpvista\matt S-1-5-21-3571944088-1297126955-1855943037-1000
```

GROUP INFORMATION

Group Name	Type	SID	Attributes
Everyone	Well-known group	S-1-1-0	Mandatory group Enabled by default Enabled group
BUILTIN\Administrators	Alias	S-1-5-32-544	Group used for deny only
BUILTIN\Users	Alias	S-1-5-32-545	Mandatory group Enabled by default Enabled group
NT AUTHORITY\INTERACTIVE	Well-known group	S-1-5-4	Mandatory group Enabled by default Enabled group
NT AUTHORITY\Authenticated Users	Well-known group	S-1-5-11	Mandatory group Enabled by default Enabled group
NT AUTHORITY\This Organization	Well-known group	S-1-5-15	Mandatory group Enabled by default Enabled group
LOCAL	Well-known group	S-1-2-0	Mandatory group Enabled by default Enabled group
NT AUTHORITY\NTLM Authentication	Well-known group	S-1-5-64-10	Mandatory group Enabled by default Enabled group
Mandatory Label\Low Mandatory Level	Unknown SID type	S-1-16-4096	Mandatory group Enabled by default Enabled group

PRIVILEGES INFORMATION

```
Privilege Name                     Description                    State
SeChangeNotifyPrivilege            Bypass traverse checking       Enabled
SeTimeZonePrivilege                Change the time zone           Disabled
SeIncreaseWorkingSetPrivilege      Increase a process working set Disabled
SeUndockPrivilege                  Remove  computer  from docking Disabled
                                   station
SeShutdownPrivilege                Shut down the system           Disabled
```

B. Protected Administrator, LUA (Medium Integrity)

USER INFORMATION

```
User Name    SID
hpvista\matt S-1-5-21-3571944088-1297126955-1855943037-1000
```

GROUP INFORMATION

Group Name	Type	SID	Attributes
Everyone	Well-known group	S-1-1-0	Mandatory group Enabled by default Enabled group

BUILTIN\Administrators	Alias	S-1-5-32-544	Group used for deny only
BUILTIN\Users	Alias	S-1-5-32-545	Mandatory group Enabled by default Enabled group
NT AUTHORITY\INTERACTIVE	Well-known group	S-1-5-4	Mandatory group Enabled by default Enabled group
NT AUTHORITY\Authenticated Users	Well-known group	S-1-5-11	Mandatory group Enabled by default Enabled group
NT AUTHORITY\This Organization	Well-known group	S-1-5-15	Mandatory group Enabled by default Enabled group
LOCAL	Well-known group	S-1-2-0	Mandatory group Enabled by default Enabled group
NT AUTHORITY\NTLM Authentication	Well-known group	S-1-5-64-10	Mandatory group Enabled by default Enabled group
Mandatory Label\Low Mandatory Level	Unknown SID type	S-1-16-8192	Mandatory group Enabled by default Enabled group

PRIVILEGES INFORMATION

```
Privilege Name                 Description                          State
SeChangeNotifyPrivilege        Bypass traverse checking             Enabled
SeTimeZonePrivilege            Change the time zone                 Disabled
SeIncreaseWorkingSetPrivilege  Increase a process working set       Disabled
SeUndockPrivilege              Remove computer from docking station Disabled
SeShutdownPrivilege            Shut down the system                 Disabled
```

C. Protected Administrator, Unrestricted (High Integrity)

USER INFORMATION

```
User Name     SID
hpvista\matt S-1-5-21-3571944088-1297126955-1855943037-1000
```

GROUP INFORMATION

Group Name	Type	SID	Attributes
Everyone	Well-known group	S-1-1-0	Mandatory group Enabled by default Enabled group
BUILTIN\Administrators	Alias	S-1-5-32-544	Mandatory group Enabled by default Enabled group Group owner
BUILTIN\Users	Alias	S-1-5-32-545	Mandatory group Enabled by default Enabled group
NT AUTHORITY\INTERACTIVE	Well-known group	S-1-5-4	Mandatory group Enabled by default Enabled group
NT AUTHORITY\Authenticated Users	Well-known group	S-1-5-11	Mandatory group Enabled by default Enabled group
NT AUTHORITY\This Organization	Well-known group	S-1-5-15	Mandatory group Enabled by default Enabled group
LOCAL	Well-known group	S-1-2-0	Mandatory group Enabled by default Enabled group
NT AUTHORITY\NTLM Authentication	Well-known group	S-1-5-64-10	Mandatory group Enabled by default Enabled group

Mandatory Label\ High Mandatory Level	Unknown SID type	S-1-16-12288	Mandatory group Enabled by default Enabled group

PRIVILEGES INFORMATION

```
Privilege Name                     Description                                      State
SeChangeNotifyPrivilege            Bypass traverse checking                         Enabled
SeSecurityPrivilege                Manage auditing and security log                 Disabled
SeBackupPrivilege                  Back up files and directories                    Disabled
SeRestorePrivilege                 Restore files and directories                    Disabled
SeSystemtimePrivilege              Change the system time                           Disabled
SeShutdownPrivilege                Shut down the system                             Disabled
SeRemoteShutdownPrivilege          Force shutdown from a remote system              Disabled
SeTakeOwnershipPrivilege           Take ownership of files or other objects         Disabled
SeDebugPrivilege                   Debug programs                                   Disabled
SeSystemEnvironmentPrivilege       Modify firmware environment values               Disabled
SeSystemProfilePrivilege           Profile system performance                       Disabled
SeProfileSingleProcessPrivilege    Profile single process                           Disabled
SeIncreaseBasePriorityPrivilege    Increase scheduling priority                     Disabled
SeLoadDriverPrivilege              Load and unload device drivers                   Disabled
SeCreatePagefilePrivilege          Create a pagefile                                Disabled
SeIncreaseQuotaPrivilege           Adjust memory quotas for a process               Disabled
SeUndockPrivilege                  Remove computer from docking station             Disabled
SeManageVolumePrivilege            Perform volume maintenance tasks                 Disabled
SeImpersonatePrivilege             Impersonate a client after authentication        Enabled
SeCreateGlobalPrivilege            Create global objects                            Enabled
SeCreateSymbolicLinkPrivilege      Create symbolic links                            Disabled
SeIncreaseWorkingSetPrivilege      Increase a process working set                   Disabled
SeTimeZonePrivilege                Change the time zone                             Disabled
```

D. LocalSytsem

USER INFORMATION

```
User Name          SID
nt authority\system S-1-5-18
```

GROUP INFORMATION

Group Name	Type	SID	Attributes
Mandatory Label\UI Access Mandatory Level	Unknown SID type	S-1-16-16640	
Everyone	Well-known group	S-1-1-0	Mandatory group Enabled by default Enabled group
BUILTIN\Users	Alias	S-1-5-32-545	Mandatory group Enabled by default Enabled group
NT AUTHORITY\SERVICE	Well-known group	S-1-5-6	Mandatory group Enabled by default Enabled group
NT AUTHORITY\Authenticated Users	Well-known group	S-1-5-11	Mandatory group Enabled by default Enabled group
NT AUTHORITY\This Organization	Well-known group	S-1-5-15	Mandatory group Enabled by default Enabled group
Unknown SID type	S-1-5-80-957450137-3151016590-2548878560-3726258338-3930544608		Enabled by default Enabled group Group owner
Unknown SID type	S-1-5-80-917953661-2020045820-2727011118-2260243830-4032185929		Enabled by default Group owner
Unknown SID type	S-1-5-80-2006800713-1441093265-		Enabled by default

		249754844-3404434343-1444102779	Enabled group Group owner
Unknown SID type		S-1-5-80-2898649604-2335086160-1904548223-3761738420-3855444835	Enabled by default Enabled group Group owner
Unknown SID type		S-1-5-80-4022436659-1090538466-1613889075-870485073-3428993833	Enabled by default Group owner
Unknown SID type		S-1-5-80-2009329905-444645132-2728249442-922493431-93864177	Enabled by default Group owner
LOCAL	Well-known group	S-1-2-0	Mandatory group Enabled by default Enabled group
BUILTIN\Administrators	Alias	S-1-5-32-544	Mandatory group Enabled by default Enabled group Group owner
Mandatory Label\System Mandatory Level	Unknown SID type	S-1-16-16384	Enabled by default Enabled Group Group Owner

PRIVILEGES INFORMATION

```
Privilege Name           Description                      State
SeTcbPrivilege           Act as part of the operating system Enabled
SeChangeNotifyPrivilege  Bypass traverse checking         Enabled
```

E. RunAsAdmin privileged COM Objects

```
*** Privileged COM under HKLM\SOFTWARE\Classes\CLSID (uses CoCreateAsAdmin)
HKLM\SOFTWARE\Classes\CLSID\{08d450b7-f7e5-4424-8229-11888adb7c14}
      InProcServer32 = %SystemRoot%\system32\fontext.dll
      AppID = {642ef9d6-48a5-476b-919a-a507cfd02c0f}
HKLM\SOFTWARE\Classes\CLSID\{1138506a-b949-46a7-b6c0-ee26499fdeaf}
      InProcServer32 = %SystemRoot%\system32\wucltux.dll
      AppID = {f62fdd2e-66d2-423b-9a04-f71ea00f892a}
HKLM\SOFTWARE\Classes\CLSID\{26FE7361-BD5A-4DCB-B309-C6F42DDE661C}
      LocalServer32 = "%ProgramFiles%\Internet Explorer\IEInstal.exe"
      AppID = {7B29F495-0F55-49F7-8885-9E8A22CE3829}
HKLM\SOFTWARE\Classes\CLSID\{304CE942-6E39-40D8-943A-B913C40C9CD4}
      InProcServer32 = c:\Windows\system32\wfapi.dll
      AppID = {304CE942-6E39-40D8-943A-B913C40C9CD4}
      ProgID = HNetCfg.FwMgr
HKLM\SOFTWARE\Classes\CLSID\{33E5987B-CA8A-4a8a-921A-8AC16A1676EB}
      InProcServer32 = %SystemRoot%\System32\shpafact.dll
      AppID = {33E5987B-CA8A-4a8a-921A-8AC16A1676EB}
HKLM\SOFTWARE\Classes\CLSID\{375C3A49-8654-49C6-BD32-7E7FE88509B4}
      InProcServer32 = %programfiles%\AdhocMeetings\WinCollabElev.dll
      AppID = {ADBC18BB-3226-4A5F-8976-CC0ECB8C2D13}
      ProgID = WinCollabElev.Elev.1
HKLM\SOFTWARE\Classes\CLSID\{3ad05575-8857-4850-9277-11b85bdb8e09}
      InProcServer32 = %SystemRoot%\system32\shell32.dll
      AppID = {3ad05575-8857-4850-9277-11b85bdb8e09}
HKLM\SOFTWARE\Classes\CLSID\{49F371E1-8C5C-4d9c-9A3B-54A6827F513C}
      InProcServer32 = ntshrui.dll
HKLM\SOFTWARE\Classes\CLSID\{4BC67F23-D805-4384-BCA3-6F1EDFF50E2C}
      InProcServer32 = c:\Windows\system32\wercplsupport.dll
      AppID = {4BC67F23-D805-4384-BCA3-6F1EDFF50E2C}
      ProgID = ERCLuaElevationHelper
HKLM\SOFTWARE\Classes\CLSID\{514B5E31-5596-422F-BE58-D804464683B5}
      InProcServer32 = intl.cpl
      AppID = {514B5E31-5596-422F-BE58-D804464683B5}
HKLM\SOFTWARE\Classes\CLSID\{6311429E-2F1A-4777-880F-C7289FD10169}
      InProcServer32 = ntshrui.dll
HKLM\SOFTWARE\Classes\CLSID\{7007ACD1-3202-11D1-AAD2-00805FC1270E}
      InProcServer32 = %SystemRoot%\System32\netshell.dll
      AppID = {7007ACD1-3202-11D1-AAD2-00805FC1270E}
HKLM\SOFTWARE\Classes\CLSID\{71B804C5-5577-471D-8FE5-C4A45B654EB8}
```

```
        InProcServer32 = %SystemRoot%\System32\AuxiliaryDisplayCpl.dll
        AppID = {71B804C5-5577-471D-8FE5-C4A45B654EB8}
HKLM\SOFTWARE\Classes\CLSID\{72A7994A-3092-4054-B6BE-08FF81AEEFFC}
        InProcServer32 = %SystemRoot%\System32\shpafact.dll
        AppID = {72A7994A-3092-4054-B6BE-08FF81AEEFFC}
HKLM\SOFTWARE\Classes\CLSID\{77F419AA-771A-45ff-AC66-7567FA3243D3}
        InProcServer32 = ntshrui.dll
HKLM\SOFTWARE\Classes\CLSID\{86d5eb8a-859f-4c7b-a76b-2bd819b7a850}
        InProcServer32 = %SystemRoot%\System32\shpafact.dll
        AppID = {86d5eb8a-859f-4c7b-a76b-2bd819b7a850}
HKLM\SOFTWARE\Classes\CLSID\{8c2db90a-6c3d-48fa-a571-0be2836c630c}
        InProcServer32 = %SystemRoot%\System32\shpafact.dll
HKLM\SOFTWARE\Classes\CLSID\{9df523b0-a6c0-4ea9-b5f1-f4565c3ac8b8}
        InProcServer32 = timedate.cpl
        AppID = {9df523b0-a6c0-4ea9-b5f1-f4565c3ac8b8}
HKLM\SOFTWARE\Classes\CLSID\{a036417d-768d-4566-8be4-5f5e1268fa9f}
        InProcServer32 = %SystemRoot%\System32\ntshrui.dll
        AppID = {a036417d-768d-4566-8be4-5f5e1268fa9f}
HKLM\SOFTWARE\Classes\CLSID\{A0ADD4EC-5BD3-4f70-A47B-07797A45C635}
        InProcServer32 = %SystemRoot%\System32\cscui.dll
        AppID = {A0ADD4EC-5BD3-4f70-A47B-07797A45C635}
HKLM\SOFTWARE\Classes\CLSID\{A2D75874-6750-4931-94C1-C99D3BC9D0C7}
        InProcServer32 = %ProgramFiles%\Windows Defender\MsMpCom.dll
        AppID = {A79DB36D-6218-48e6-9EC9-DCBA9A39BF0F}
HKLM\SOFTWARE\Classes\CLSID\{A3BB0AD5-ECA3-4A81-B2CB-15FD8349D400}
        LocalServer32 = %SystemRoot%\System32\SLLUA.exe
        AppID = {4C4BB7A4-0D3C-4601-A9C4-325AFB9F77BB}
        ProgID = SLLUA.SLLUAObject.1
HKLM\SOFTWARE\Classes\CLSID\{A7A63E5C-3877-4840-8727-C1EA9D7A4D50}
        InProcServer32 = %SystemRoot%\System32\fveui.dll
        AppID = {A7A63E5C-3877-4840-8727-C1EA9D7A4D50}
HKLM\SOFTWARE\Classes\CLSID\{afb8cfa2-6d7b-4108-9202-cc08d7222dc9}
        InProcServer32 = %SystemRoot%\system32\shell32.dll
        AppID = {afb8cfa2-6d7b-4108-9202-cc08d7222dc9}
HKLM\SOFTWARE\Classes\CLSID\{BCEA735B-4DAC-4B71-9C47-1D560AFD2A9B}
        InProcServer32 = DfsShlEx.dll
        AppID = {BCEA735B-4DAC-4B71-9C47-1D560AFD2A9B}
        ProgID = DfsShell.DfsShellAdmin.1
HKLM\SOFTWARE\Classes\CLSID\{C529C7EF-A3AF-45F2-8A47-767B33AA5CC0}
        InProcServer32 = %SystemRoot%\system32\ndfapi.dll
        AppID = {F3D3AA8D-EF96-4470-848E-BD70B803047A}
        ProgID = ndfapi.NDFAPI.1
HKLM\SOFTWARE\Classes\CLSID\{cee8ccc9-4f6b-4469-a235-5a22869eef03}
        InProcServer32 = PNPXAssoc.dll
        AppID = {cee8ccc9-4f6b-4469-a235-5a22869eef03}
HKLM\SOFTWARE\Classes\CLSID\{D3667F1E-CCB8-4A69-99DF-59A2B2A6753F}
        InProcServer32 = %SystemRoot%\System32\AuxiliaryDisplayCpl.dll
        AppID = {D3667F1E-CCB8-4A69-99DF-59A2B2A6753F}
HKLM\SOFTWARE\Classes\CLSID\{e6f59608-8aa2-4dbe-a651-c2f6585e4f30}
        InProcServer32 = %SystemRoot%\System32\shpafact.dll
        AppID = {e6f59608-8aa2-4dbe-a651-c2f6585e4f30}
HKLM\SOFTWARE\Classes\CLSID\{E9495B87-D950-4ab5-87A5-FF6D70BF3E90}
        InProcServer32 = wscui.cpl
        AppID = {E9495B87-D950-4ab5-87A5-FF6D70BF3E90}
HKLM\SOFTWARE\Classes\CLSID\{edb5f444-cb8d-445a-a523-ec5ab6ea33c7}
        InProcServer32 = ntshrui.dll
        AppID = {edb5f444-cb8d-445a-a523-ec5ab6ea33c7}

*** Privileged COM under HKCU\SOFTWARE\Classes\CLSID (uses CoCreateAsAdmin)

None
```

F. File Extensions Excluded from Virtualization

acm	cer	csh	hta	maf	maw	mst	pst	url	xsd
ade	chm	dll	ime	mag	mda	mui	reg	vbe	xsl
adp	clb	drv	inf	mam	mdb	nls	scf	vbs	

app	cmd	dtd	ins	man	mde	ocx	scr	vsmacros
asa	cnt	exe	isp	maq	mdt	ops	sct	vss
asp	cnv	fon	its	mar	mdw	pal	shb	vst
aspx	com	fxp	jse	mas	mdz	pcd	shs	vsw
bas	cpl	grp	ksh	mat	msc	pif	sys	wsc
bat	cpx	hlp	lnk	mau	msi	prf	tlb	wsf
bin	crt	hls	mad	mav	msp	prg	tsp	wsh

G. Renamed System Calls (between XP SP2 and Vista)

From	To
ZwAddBootEntry	NtAddDriverEntry
ZwClose	NtClose
ZwCreateKey	NtCreateKey
ZwCreatePagingFile	NtCreatePagingFile
ZwDeleteDriverEntry	NtDeleteDriverEntry
ZwEnumerateBootEntries	NtEnumerateDriverEntries
ZwEnumerateKey	NtEnumerateKey
ZwEnumerateValueKey	NtEnumerateValueKey
ZwModifyBootEntry	NtModifyDriverEntry
ZwOpenKey	NtOpenKey
ZwOpenSession	NtOpenSession
ZwQueryBootEntryOrder	NtQueryDriverEntryOrder
ZwQueryKey	NtQueryKey
ZwQueryValueKey	NtQueryValueKey
ZwSetBootEntryOrder (from)	NtSetDriverEntryOrder

H. Added System Calls (between Windows XP SP2 and Windows Vista)

NtAlpcAcceptConnectPort	NtDereferenceEnlistmentKey	NtRecoverTransactionManager
NtAlpcCancelMessage	NtFlushProcessWriteBuffers	NtReferenceEnlistmentKey
NtAlpcConnectPort	NtFreezeRegistry	NtRegisterProtocolAddressInformation
NtAlpcCreatePort	NtFreezeTransactions	NtReleaseWorkerFactoryWorker
NtAlpcCreateResourceReserve	NtGetCurrentProcessorNumber	NtRenameTransactionManager
NtAlpcCreateSectionView	NtGetNextProcess	NtRollbackComplete
NtAlpcCreateSecurityContext	NtGetNextThread	NtRollbackEnlistment
NtAlpcDeletePortSection	NtGetNlsSectionPtr	NtRollbackTransaction
NtAlpcDeleteResourceReserve	NtGetNotificationResourceManager	NtRollforwardTransactionManager
NtAlpcDeleteSectionView	NtInitializeNlsFiles	NtSavepointComplete
NtAlpcDeleteSecurityContext	NtListTransactions	NtSavepointTransaction
NtAlpcDisconnectPort	NtMapCMFModule	NtSetInformationEnlistment
NtAlpcImpersonateClientOfPort	NtMarshallTransaction	NtSetInformationResourceManager
NtAlpcQueryInformation	NtOpenEnlistment	NtSetInformationTransaction
NtAlpcReceiveBatchMessages	NtOpenResourceManager	NtSetInformationWorkerFactory
NtAlpcSendBatchMessage	NtOpenTransaction	NtSetSystemPowerState
NtAlpcSendWaitReceivePort	NtPrePrepareComplete	NtShutdownWorkerFactory
NtAlpcSetInformation	NtPrePrepareEnlistment	NtSinglePhaseReject
NtApphelpCacheControl	NtPrepareComplete	NtStartTm
NtCancelSynchronousIoFile	NtPrepareEnlistment	NtThawRegistry
NtClearAllSavepointsTransaction	NtPropagationComplete	NtThawTransactions
NtClearSavepointTransaction	NtPropagationFailed	NtTraceControl
NtCommitComplete	NtPullTransaction	NtWaitForWorkViaWorkerFactory
NtCommitEnlistment	NtQueryInformationEnlistment	NtWorkerFactoryWorkerReady

```
NtCommitTransaction              NtQueryInformationResourceManager

NtCreateEnlistment               NtQueryInformationTransaction

NtCreateResourceManager          NtQueryInformationWorkerFactory

NtCreateTransaction              NtQueryLicenseValue

NtCreateUserProcess              NtReadOnlyEnlistment

NtCreateWorkerFactory            NtRecoverResourceManager
```

I. Removed System Calls (between Windows XP SP2 and Windows Vista)

```
NtCloseObjectAuditAlarm
NtCreateKeyedEvent
NtOpenKeyedEvent
NtSetSystemEnvironmentValueEx
```